The Search for the Laws of Automatic Random Testing

Carlo A. Furia* · Bertrand Meyer* · Manuel Oriol†‡ · Andrey Tikhomirov§ · Yi Wei*

*Chair of Software Engineering, ETH Zurich
Zurich, Switzerland
{firstname.lastname@inf.ethz.ch}

†ABB Corporate Research, Industrial Software Systems
Baden-Dättwil, Switzerland
manuel.oriol@ch.abb.com

‡Dept. of Computer Science, University of York
York, UK

§Software Engineering Laboratory, ITMO
St. Petersburg, Russia
and.tikhomirov@gmail.com

ABSTRACT

Can one estimate the number of remaining faults in a software system? A credible estimation technique would be immensely useful to project managers as well as customers. It would also be of theoretical interest, as a general law of software engineering. We investigate possible answers in the context of automated random testing, a method that is increasingly accepted as an effective way to discover faults. Our experimental results, derived from best-fit analysis of a variety of mathematical functions, based on a large number of automated tests of library code equipped with automated oracles in the form of contracts, suggest a poly-logarithmic law. Although further confirmation remains necessary on different code bases and testing techniques, we argue that understanding the laws of testing may bring significant benefits for estimating the number of detectable faults and comparing different projects and practices.

1. INTRODUCTION

A scientific discipline is characterized by general laws, such as Maxwell's equations. Where the topics of discourse involve human phenomena, and belong to engineering rather than science, the laws cannot be absolute truths comparable to the laws of nature, and often involve a probabilistic element; but they should still describe properties and constraints that govern all or almost all instances of the phenomena under consideration.

Software engineering is particularly in need of such laws. Some useful ones have already been identified, in areas such as project management; an example [5, 12] is the observation, first expressed by Barry Boehm on the basis of his study of a large database of projects, that every software project has a *nominal* cost, deducible from the project's overall characteristics, from which it is not possible to deviate – whether by giving the project more time or by including more developers – by more than about 25%.

One area in which such a general law would be of particular interest is testing. It has been well known since at least Lehmann and Belady's seminal work on faults ("bugs") in successive IBM OS 360 releases [4] that if a project tracks faults in a sufficiently diligent way the evolution of the number of faults in successive releases follows regular patterns. Is it possible to turn this general, informal observation into a precise *law of software testing*, on which project managers and developers could rely to estimate, from the number of faults uncovered so far through testing, the number of *detectable* faults in an individual module or an entire system? This is the question that we set out to explore, and on which we are reporting our first results.

1.1 Expected benefits

Were it available, and backed by experimental evidence covering a broad enough base of projects and environments to make it credible, a law describing the rate of fault detection would be of considerable interest to the industry.

An immediate benefit would be to help project managers answer one of the most common and also toughest questions they face: *can we ship yet*? A release should be shipped early enough to avoid missing a market opportunity or being scooped by the competition; but not too early if this means that so many faults remain as to provoke an adverse reaction from the market. Another application, for individual developers, is to estimate the amount of testing that remains necessary for a module or a subsystem. Yet another benefit would be to help assess how a project's or organization's fault patterns differ, for better or worse, from average industry practices. Such an assessment would be particularly appropriate in organizations applying strict guidelines, for example as part of CMMI [8]. Also of interest is the purely intellectual benefit of gaining insights into the nature of software development, in the form of a general law of software engineering that describes fault patterns and might help in the effort to avoid faults in the first place.

1.2 Automatic testing

In the present work, we set out to determine through empirical study if a general law of testing exists, predicting the number of detectable faults. The context of the study is *automatic* testing, in which test cases are not manually prepared by humans but generated automatically through appropriate algorithms. More specifically, we rely on automated *random* testing, where these algorithms use partly random techniques. Once dismissed as ineffective [14], random testing has shown itself, in recent years, to be a vi-

able technique; such tools as AutoTest [13] for Eiffel and Randoop [19] and YETI [17] for Java and .NET, which apply automatic random testing, succeed by the criterion that matters most in practice: finding faults in programs, including released libraries and systems.

With automated testing, the user selects a number of modules – in object-oriented programming, classes – for testing; the testing tool then creates objects (class instances) and performs method calls on them with arguments selected according to a strategy that involves a random component. It then records any abnormal behavior (failure) that may be signaling a fault in the software.

The work described here relies on automated random testing as just described. This approach has the advantage of simplicity, and of not relying on hard-to-control human actions. Another advantage is reproducibility (up to the variation caused by the choice of seeds in random number generation). It also has the disadvantage of limiting the generalizability of our findings (Section 3.2); but it is hardly surprising that a general empirical understanding of testing is likely to require much more investigation and varied experiments.

The paper's main experiments involve software equipped with *contracts* (specification elements embedded in the software and subject to run-time evaluation, such as pre- and postconditions and class invariants); in such a context, the testing process focuses on generating failures due to contract violations. Since contracts are meant to express the *specification* of the methods and classes being tested, they provide testing *oracles* [23] to reliably identify *faults* (as opposed to mere failures). The bulk of our experiments (Section 2.2) target code with contracts and studies the evolution of found faults over time. Therefore, the goal is to obtain experimentally a general law $\mathcal{F}(t)$, where t is the testing time, measured in number of drawn test cases and $\mathcal{F}(t)$ is the number of unique faults uncovered by the tests up to time t.

If contracts are not available, a fully automated approach must rely on failures such as arithmetic overflow or running out of memory. In these cases, a methodological issue arises in practice since automated testing finds failures rather than faults; obtaining $\mathcal{F}$ requires a sound policy to determine whether two given failures relate to the same fault. We hint at some ways to address this problem in Section 2.3.

Extended version. More detailed data and graphs are available in an extended version of the paper at:

Appendix: Section 6 of this paper.

2. EXPERIMENTS

The experiments run repeated random testing sessions on Eiffel classes (Section 2.2) with AutoTest, and on Java classes (Section 2.3) with YETI. Each testing *session* is characterized by:

- c: the tested *class*;
- T: the number of rounds in the simulation, that is the number of *test cases* (valid and invalid) drawn;
- $\varphi : [0..\mathrm{T}] \to \mathbb{N}$: the function counting the cumulative number of unique failures or faults after each round.

The dataset D^c for a class c consists of a collection of counting functions $\varphi_1, \varphi_2, \ldots, \varphi_{\mathrm{S}^c}$, where S^c is the total number of testing sessions on c. In our experiments, all source code was tested "as is", without injecting any artificial error or modifying it in any way.

Given a dataset D^c, consider the mean function of the fault count:

$$\phi^c_{\mathrm{mean}}(k) \quad = \quad \frac{1}{\mathrm{S}^c} \sum_{1 \le i \le \mathrm{S}^c} \varphi_i(k)$$

as well as the median $\phi^c_{\mathrm{median}}(k)$. We considered nine model functions, described in Section 2.1, suggested by various sources, and fit each model function to each of $\phi^c_{\mathrm{mean}}(k)$ and $\phi^c_{\mathrm{median}}(k)$ from the experimental data. We used Matlab 2011 for all data analysis and fitting computations. The following sections illustrate the main results.

2.1 Model functions

We consider a number of "reasonable" fitting functions, suggested either by intuition or by the theoretical model discussed in Section 3.1. Some of the functions are special cases of other functions; nonetheless, it is advisable to try to fit both – special and general case – because the performance of the fitting algorithms may be sensitive to the form in which a function is presented. In all the examples, lower- and upper-case names other than the argument x denote parameters to be instantiated by fitting.

Following an original intuition of analogy with biological phenomena – suggested in related work [16] – we consider the Michaelis-Menten equation:

$$\Phi_1(x) \quad = \quad \frac{ax}{x + B} \tag{1}$$

as well as its generalizations into a rational function of third degree:

$$\Phi_2(x) \quad = \quad \frac{ax^3 + bx^2 + cx + d}{Ax^3 + Bx^2 + Cx + D} \tag{2}$$

and a rational function of arbitrary degree:

$$\Phi_3(x) \quad = \quad \frac{ax^b + c}{Ax^B + C} \tag{3}$$

The model analyzed in Section 3.1 suggests including functional forms similar to polynomials, as well as logarithms and exponentials. Hence, we consider logarithms to any power:

$$\Phi_4(x) \quad = \quad a \log^b(x + 1) + c \tag{4}$$

and poly-logarithms of third degree:

$$\Phi_5(x) \quad = \quad a \log^3(x+1) + b \log^2(x+1) + c \log(x+1) + d \tag{5}$$

where the translation coefficient +1 is needed because the fault function ϕ is such that $\phi(0) = 0$, but $\log(0) = -\infty$. The base of the logarithm is immaterial because the multiplicative parameters can accommodate any. We also consider exponentials of powers:

$$\Phi_6(x) \quad = \quad ab^{x^{1/c}} + d \tag{6}$$

Polynomials up to degree three are covered by Φ_2, but we still consider some special cases explicitly; third degree:

$$\Phi_7(x) \quad = \quad ax^3 + bx^2 + cx + d \tag{7}$$

arbitrary degree:

$$\Phi_8(x) \quad = \quad ax^b + c \tag{8}$$

and third degree with negative powers:

$$\Phi_9(x) \quad = \quad ax^{-3} + bx^{-2} + cx^{-1} + d \tag{9}$$

2.2 Experiments with Eiffel code

Experiments with Eiffel used AutoTest [13]; they targeted 42 Eiffel classes, from the widely used open-source data structure libraries EiffelBase [10] and Gobo [11]. Table 1 reports the statistics about the testing sessions with AutoTest on these classes; for each class, the table lists the number of testing sessions (S), the test cases sampled per session (T), the maximum number of unique faults found (F, corresponding to unique contract violations), the mean sampled standard deviation and skewness ($\mathbb{E}[\sigma], \mathbb{E}[\gamma]$) of the faults found across the class's multiple testing sessions, and the mean and standard deviation ($\mathbb{E}[\Delta], \sigma[\Delta]$) of the new faults found with every test case sampled (thus, for example, $\mathbb{E}[\Delta] = 10^{-3}$ means that a new fault is found every thousand test cases on average). The bottom rows display mean, median, and standard deviation of the values in each column. The data was collected according to the suggested guidelines for empirical evaluation of randomized algorithms [2]; in particular, the very large number of drawn test cases makes the averaged data robust with respect to statistical fluctuations.

2.2.1 Fitting results

Table 2 reports the result of fitting the models Φ_1–Φ_9 on the mean fault count $\phi^c_{\text{mean}}(k)$ curves; for each class, column BEST FIT RANKING ranks the Φ_i's from the best fitting to the worst, according to the coefficient of determination R^2 (the higher, the better), and the root mean squared error RMSE (the smaller, the better); the rankings according to the two measures agreed in all experiments. The other columns report the R^2 and RMSE scores of the best fit, and the absolute value of the difference between such scores for the best fit and the same scores for Φ_5. The data shows that Φ_5 emerges as consistently better than the other functions; as the bottom of the table summarizes, Φ_5 is the best fit in 62% and one of the top two fits in 90% of the cases; when it is not the best, it fares within 1% of the best in terms of scores. A Wilcoxon signed-rank test confirms that the observed differences between Φ_5 and the other functions are highly statistically significant: for example, a comparison of the R^2 values indicates that the values for Φ_5 are different (smaller) with high probability ($7.14 \cdot 10^{-7} < p < 1.65 \cdot 10^{-8}$) and large effect size (between 0.54 and 0.62, computed using the Z-statistics). The same data with respect to the median curves is qualitatively the same as with the mean; hence we do not report it in detail.

In a few cases, models other than Φ_5 achieve a higher fitting score even if visual inspection of the curves shows that it is obviously unsatisfactory. The reasons for this behavior are arguably due to numerical errors; there are, however, a few cases of fits with high scores but visibly unsatisfactory, whose fitting process converged without Matlab signaling any numerical approximation problem. In any case, visual inspection confirms that Φ_5 is a visibly proper fit to φ in all cases.

2.2.2 Poly-logarithmic fits

Once acknowledged that the evidence points towards a poly-logarithmic law, it remains the question of which polynomial degree achieves the best fit. Consider seven poly-logarithmic model functions $\mathcal{L}_k$, $1 \leq k \leq 7$, of various degrees. $\mathcal{L}_1$–$\mathcal{L}_5$ have degree equal to the number of their nonzero coefficients plus one:

$$\mathcal{L}_k(x) \quad = \quad c_0 + \sum_{1 \leq j \leq k} c_j \log^j(x+1), 1 \leq k \leq 5 \qquad (10)$$

$\mathcal{L}_6$ and $\mathcal{L}_7$ are, instead, binomials whose degree is a parameter; namely, $\mathcal{L}_6$ equals Φ_4 and:

$$\mathcal{L}_7(x) \quad = \quad a \log^{1/b}(x+1) + c \qquad (11)$$

The results show that it is always the case that $\mathcal{L}_5 > \mathcal{L}_4 > \mathcal{L}_3 > \mathcal{L}_2 > \mathcal{L}_1$, that is the higher the degree the better the fit; in hindsight, this is an unsurprising consequence of the fact that models of higher degree offer more degrees of freedom, hence they are more flexible. $\mathcal{L}_3 = \Phi_5$, however, still fares quite well on average; the few cases where its performance is as much as 10% worse than $\mathcal{L}_5$ correspond to more irregular curves with fewer faults and hence fewer new datapoints, such as for class *ARRAYED_QUEUE*. Finally, $\mathcal{L}_6$ and $\mathcal{L}_7$ occasionally rank third; whenever this happens, however, the difference with the best (and with Φ_5) is negligible. In all, Φ_5 seems to be a reasonable compromise between flexibility and economy, but poly-logarithmic functions of higher order could be considered when useful. Notice that, even if in principle we could obtain enough degrees of freedom with any function of arbitrarily high degree, only poly-logarithmic works for reasonable degrees; for example, polynomials of fifth degree (which generalize Φ_7) never provide good fits.

2.3 Experiments with Java code

With the goal of confirming (or invalidating) the results of the Eiffel experiments, we used YETI to test 9 Java classes from *java.util* and 29 classes from *java.lang*. For lack of space, we only present a summary of the results and highlight the differences in comparison with Eiffel and the questions about the comparison that remain open.

Unlike in Eiffel, where the code has contracts, interpreting a Java failure to know whether it is a "real" fault cannot be done in a fully automated way in normal Java code without contracts. To address this point at least in part, YETI collects all the exceptions triggered during random testing and considers the following exceptions as *not* provoked by a fault, but merely accidents of the testing process: (1) declared exceptions, including *RuntimeException*, as client code knows that they may be thrown and may even be part of a code pattern; (2) *InvalidArgumentException*, comparable to precondition violations. This still falls short of a complete identification of real faults, but it helps to reduce the number of spurious failures.

Given this filtering performed by YETI, the experiments with the 38 Java classes are in two parts. The first part targets the 9 *java.util* classes and 2 classes from *java.lang*, and analyzes all unique *failures* filtered by YETI as described above (two failures are the same if they generate the same stack exception trace). The second part targets the 29 classes from *java.lang* and only reports failures manually pruned by discarding all failures that reflect behavior compatible with the informal documentation (for example, division by zero when the informal API documentation requires an argument to be non-zero). In this case, we get to a fairly reasonable approximation of real *faults*; hence we will refer to the first part of Java experiments as "failures" and to the second as "faults".

Table 1: Eiffel classes statistics.

Class	S	T	F	$\mathbb{E}[\sigma]$	$\mathbb{E}[\gamma]$	$\mathbb{E}[\Delta]$	$\sigma[\Delta]$
ACTIVE_LIST	11	2.14E+06	18	1.0553	-4.88E+00	8.39E-04	2.27E-19
ARRAY	11	1.58E+06	69	5.0972	-3.36E+00	4.24E-03	9.02E-05
ARRAYED_CIRCULAR	15	7.77E+05	52	5.2882	-2.09E+00	6.12E-03	2.75E-04
ARRAYED_LIST	14	1.25E+06	24	1.4279	-4.06E+00	1.70E-03	9.89E-05
ARRAYED_QUEUE	12	1.99E+06	6	0.3742	-2.00E+01	2.56E-04	1.45E-05
ARRAYED_SET	15	2.69E+06	15	1.0823	-3.72E+00	5.34E-04	2.35E-05
BINARY_TREE	13	2.25E+06	40	5.3739	-2.14E+00	1.66E-03	7.56E-05
BOUNDED_QUEUE	14	2.34E+06	5	0.3135	-1.46E+01	2.14E-04	0.00E+00
COMPACT_CURSOR_TREE	14	5.56E+05	67	5.6622	-2.47E+00	1.13E-02	3.35E-04
DS_ARRAYED_LIST	12	2.07E+06	91	7.0082	-3.69E+00	4.29E-03	6.65E-05
DS_AVL_TREE	15	5.10E+05	16	2.0347	-1.82E+00	2.53E-03	2.02E-04
DS_BILINKED_LIST	14	2.70E+06	30	2.2035	-3.77E+00	8.90E-04	9.43E-05
DS_BINARY_SEARCH_TREE	11	8.95E+05	15	1.8224	-1.93E+00	1.51E-03	9.16E-05
DS_BINARY_SEARCH_TREE_SET	12	5.96E+05	5	0.4048	-6.31E+00	5.18E-04	1.67E-04
DS_HASH_SET	14	1.86E+06	19	1.5352	-1.32E+00	6.31E-04	1.83E-04
DS_HASH_TABLE	14	1.66E+06	25	2.0158	-1.39E+00	1.07E-03	1.88E-04
DS_LEFT_LEANING_RED_BLACK_TREE	14	7.18E+05	33	2.9464	-3.45E+00	4.32E-03	1.73E-04
DS_LINKED_LIST	14	2.17E+06	94	7.7943	-2.67E+00	3.88E-03	1.88E-04
DS_LINKED_QUEUE	14	2.23E+06	9	1.3880	-4.98E-01	3.05E-04	5.89E-05
DS_LINKED_STACK	14	2.22E+06	10	1.4648	-5.39E-01	3.57E-04	6.23E-05
DS_MULTIARRAYED_HASH_SET	14	2.61E+06	22	3.7948	-7.99E-01	7.61E-04	5.60E-05
DS_RED_BLACK_TREE	15	6.62E+05	32	2.9321	-3.55E+00	4.62E-03	9.67E-05
FIXED_TREE	13	7.54E+05	49	4.9311	-2.47E+00	6.25E-03	1.67E-04
HASH_TABLE	14	2.95E+06	24	3.6472	-8.98E-01	6.59E-04	8.49E-05
LINKED_AUTOMATON	14	4.03E+06	14	0.4384	-1.74E+01	3.28E-04	1.06E-05
LINKED_CIRCULAR	15	1.50E+06	40	4.2747	-1.92E+00	2.50E-03	1.10E-04
LINKED_CURSOR_TREE	14	1.34E+06	51	4.6304	-1.75E+00	3.56E-03	1.42E-04
LINKED_DFA	13	2.72E+06	135	12.9783	-1.97E+00	4.67E-03	1.69E-04
LINKED_LIST	14	3.41E+06	21	1.4436	-2.84E+00	5.30E-04	3.89E-05
LINKED_PRIORITY_QUEUE	14	1.20E+06	23	2.1477	-2.34E+00	1.80E-03	6.35E-05
LINKED_SET	14	2.59E+06	29	2.1800	-3.45E+00	1.03E-03	4.48E-05
LINKED_TREE	15	1.68E+06	136	22.3981	-9.56E-01	7.20E-03	4.06E-04
MULTI_ARRAY_LIST	14	4.73E+05	35	3.4960	-3.22E+00	6.92E-03	3.66E-04
PART_SORTED_SET	13	3.27E+06	33	3.6535	-2.12E+00	9.60E-04	3.43E-05
PART_SORTED_TWO_WAY_LIST	13	2.61E+06	36	3.7867	-2.01E+00	1.28E-03	5.73E-05
SORTED_TWO_WAY_LIST	15	2.60E+06	34	3.3816	-2.37E+00	1.22E-03	4.64E-05
SUBSET_STRATEGY_TREE	15	3.08E+06	20	1.8972	-2.87E+00	5.84E-04	3.47E-05
TWO_WAY_CIRCULAR	13	1.53E+06	41	4.3326	-1.88E+00	2.46E-03	1.37E-04
TWO_WAY_CURSOR_TREE	13	1.35E+06	50	4.5548	-1.86E+00	3.56E-03	1.11E-04
TWO_WAY_LIST	15	2.01E+06	22	1.7119	-3.45E+00	9.61E-04	6.42E-05
TWO_WAY_SORTED_SET	14	3.09E+06	47	4.2784	-3.43E+00	1.47E-03	3.29E-05
TWO_WAY_TREE	14	2.46E+06	163	28.6749	-1.02E+00	6.27E-03	1.77E-04
Mean	14	1.93E+06	40	4.3299	-3.55E+00	2.54E-03	1.15E-04
Median	14	2.04E+06	31	3.1640	-2.42E+00	1.49E-03	9.09E-05
Stdev	1	9.04E+05	36	5.3955	4.08E+00	2.49E-03	9.59E-05

Tables 3–4 report the summary statistics about the testing sessions with YETI on all classes, with the same data as for Eiffel (see Section 2.2). The data for faults, where manual pruning eliminated the vast majority of failures as spurious, has far fewer significant datapoints. Skewness is not computable when there are no faults found, which happens for 15 classes; hence the summary statistics about skewness are immaterial (NaN stands for "Not a Number").

Table 3: Java classes tested for failures: statistics.

Summary	S	T	F	$\mathbb{E}[\sigma]$	$\mathbb{E}[\gamma]$	$\mathbb{E}[\Delta]$	$\sigma[\Delta]$
Mean	30	1.23E+05	32	3.9044	-8.02E+00	1.03E-02	1.86E-04
Median	30	1.18E+05	15	1.7799	-6.10E+00	7.16E-03	6.68E-05
Stdev		1.65E+04	29	3.8803	4.70E+00	8.86E-03	2.63E-04

Table 4: Java classes tested for faults: statistics.

Summary	S	T	F	$\mathbb{E}[\sigma]$	$\mathbb{E}[\gamma]$	$\mathbb{E}[\Delta]$	$\sigma[\Delta]$
Mean	4	4.68E+05	2	0.2356	NaN	1.73E-03	2.51E-05
Median	4	2.61E+05		0.0000	NaN	0.00E+00	0.00E+00
Stdev	1	9.45E+05	3	0.4094	NaN	2.72E-03	1.26E-04

Fitting results

We tried to fit the models $\Phi_1, \Phi_2, \Phi_4, \Phi_5$ on the mean fault count $\phi^c_{\text{mean}}(k)$ curves, for both Java failures and faults. The data is somewhat harder to fit than in the Eiffel experiments, and in fact we had to exclude the other models because they could produce converging fixes in only a fraction of the experiments. In comparison with the Eiffel data, there is a detectable overall difference: Φ_2 and Φ_1 seem to emerge as the best models, whereas Φ_5 is best only in a limited number of cases. Looking at the R^2 scores, Φ_1 and Φ_2 alternate as best model for the failure curves, and their difference is often small ($p = 0.067$ and effect size 0.398); for the fault curves, Φ_1 ranks first in the majority of classes, but its difference w.r.t. Φ_2 is statistically insignificant ($p = 0.5$ and effect size 0.095). A common phenomenon is that the majority of curves $\phi^c_{\text{mean}}(k)$ with YETI show an horizontal asymptote, which Φ_1 or Φ_2 can accommodate much better than Φ_5 can.

A closer look suggests that the differences w.r.t. the Eiffel experiments may still be reconcilable. First, in the majority of cases of failures, Φ_5 fits to within the 5% of the best model; the only exceptions are the *java. util* classes *ArrayList*, *Hashtable*, and *LinkedList* where visual inspection shows curves with a steeper initial phase that Φ_2 fits best, and where Φ_1 and Φ_5 behave similarly. For the fault experiments, the picture is more varied; but Φ_5 fits to within the 20% of the best models in all classes but three; and the quality of fitting a certain model is much more varied because of the small number of faults found in these experiments (but we excluded from these statistics the classes with no faults found). Finally, the difference between Φ_2 and Φ_5 is often statistically insignificant; the difference between Φ_1 and Φ_5 is significant ($p \simeq 0.2$) but not large for faults (effect size 0.3); in contrast, the difference between Φ_5 and Φ_4 is highly statistically significant and large ($p < 10^{-3}$ and effect size from 0.43 to 0.62). In all, further experiments are necessary

Table 2: Testing of Eiffel classes: best fits with mean.

Class	Fit Ranking	R^2 (best)	RMSE (best)	R^2 (Δ best)	RMSE (Δ best)
ACTIVE_LIST	2 5 4 1 9 6 8 3 7	0.9825	0.1304	0.0108	-0.0353
ARRAY	5 4 1 9 8 6 3 7 2	0.9894	0.5151	0.0000	0.0000
ARRAYED_CIRCULAR	5 3 4 8 1 2 9 6 7	0.9978	0.2461	0.0000	0.0000
ARRAYED_LIST	3 5 4 1 8 9 6 7 2	0.9877	0.1485	0.0202	-0.0932
ARRAYED_QUEUE	8 5 4 3 1 9 2 6 7	0.7299	0.1547	0.0027	-0.0008
ARRAYED_SET	5 4 1 2 8 9 6 3 7	0.9550	0.2078	0.0000	0.0000
BINARY_TREE	3 5 4 8 1 7 9 6 2	0.9987	0.1943	0.0030	-0.1566
BOUNDED_QUEUE	5 4 1 3 2 9 6 8 7	0.8048	0.1043	0.0000	0.0000
COMPACT_CURSOR_TREE	5 4 3 8 1 2 9 6 7	0.9830	0.7254	0.0000	0.0000
DS_ARRAYED_LIST	5 4 1 8 2 9 6 3 7	0.9860	0.8201	0.0000	0.0000
DS_AVL_TREE	5 4 8 3 1 2 9 6 7	0.9861	0.2299	0.0000	0.0000
DS_BILINKED_LIST	5 4 3 1 2 9 6 8 7	0.9947	0.1528	0.0000	0.0000
DS_BINARY_SEARCH_TREE	5 3 4 8 1 2 9 6 7	0.9913	0.1618	0.0000	0.0000
DS_BINARY_SEARCH_TREE_SET	8 4 5 3 2 1 9 6 7	0.8727	0.1145	0.0912	-0.0355
DS_HASH_SET	8 5 4 3 1 2 9 6 7	0.9782	0.2085	0.0021	-0.0097
DS_HASH_TABLE	8 5 4 3 1 9 6 7 2	0.9653	0.3536	0.0053	-0.0258
DS_LEFT_LEANING_RED_BLACK_TREE	5 4 3 1 8 9 6 7 2	0.9888	0.3015	0.0000	0.0000
DS_LINKED_LIST	5 4 8 1 2 9 6 3 7	0.9905	0.7505	0.0000	0.0000
DS_LINKED_QUEUE	8 4 5 7 3 1 9 6 2	0.9940	0.0986	0.0094	-0.0595
DS_LINKED_STACK	8 3 4 5 1 9 6 2 7	0.9914	0.1265	0.0077	-0.0480
DS_MULTIARRAYED_HASH_SET	5 8 4 3 7 1 2 9 6	0.9852	0.4465	0.0000	0.0000
DS_RED_BLACK_TREE	5 4 1 9 6 8 3 2 7	0.9915	0.2625	0.0000	0.0000
FIXED_TREE	5 4 1 8 2 9 6 3 7	0.9920	0.4335	0.0000	0.0000
HASH_TABLE	8 5 4 3 7 1 2 9 6	0.9979	0.1621	0.0010	-0.0353
LINKED_AUTOMATON	1 2 5 9 6 4 8 3 7	0.9693	0.0669	0.2517	-0.1361
LINKED_CIRCULAR	5 3 4 1 8 2 9 6 7	0.9982	0.1774	0.0000	0.0000
LINKED_CURSOR_TREE	5 8 4 3 1 9 6 7 2	0.9904	0.4413	0.0000	0.0000
LINKED_DFA	5 4 8 3 1 2 9 6 7	0.9954	0.8751	0.0000	0.0000
LINKED_LIST	3 5 4 8 1 2 9 6 7	0.9678	0.2441	0.0005	-0.0017
LINKED_PRIORITY_QUEUE	5 4 3 8 1 2 9 6 7	0.9866	0.2391	0.0000	0.0000
LINKED_SET	5 4 1 8 2 9 6 3 7	0.9749	0.3313	0.0000	0.0000
LINKED_TREE	5 4 8 3 7 1 2 9 6	0.9992	0.6490	0.0000	0.0000
MULTI_ARRAY_LIST	5 1 4 8 9 6 3 2 7	0.9951	0.2402	0.0000	0.0000
PART_SORTED_SET	5 4 3 8 1 2 9 6 7	0.9976	0.1747	0.0000	0.0000
PART_SORTED_TWO_WAY_LIST	5 4 8 1 3 2 9 6 7	0.9851	0.4510	0.0000	0.0000
SORTED_TWO_WAY_LIST	5 4 3 8 1 2 9 6 7	0.9790	0.4763	0.0000	0.0000
SUBSET_STRATEGY_TREE	5 3 4 1 8 2 9 6 7	0.9871	0.2066	0.0000	0.0000
TWO_WAY_CIRCULAR	5 3 4 8 1 2 9 6 7	0.9953	0.2898	0.0000	0.0000
TWO_WAY_CURSOR_TREE	5 4 1 2 9 8 6 3 7	0.9897	0.4469	0.0000	0.0000
TWO_WAY_LIST	5 4 1 8 2 9 6 3 7	0.9862	0.1899	0.0000	0.0000
TWO_WAY_SORTED_SET	5 4 1 8 3 2 9 6 7	0.9933	0.3427	0.0000	0.0000
TWO_WAY_TREE	4 5 8 3 7 1 2 9 6	0.9993	0.7493	0.0000	-0.0191
	69% (5 is best)		**Mean**	0.0097	-0.0156

to conclusively determine what the exact magnitude and the ultimate causes of these differences are: possible candidates are the fault density of the software tested, the details of the algorithms implemented by the testing tools (AutoTest and YETI), and the availability of contracts to detect faults.

3. DISCUSSION AND THREATS

3.1 Towards a justification for the law

Arcuri et al. [3] model random testing as a *coupon collector problem* [22]: "Consider a box which contains N types of numerous objects. An object in the box is repeatedly sampled on a random basis. Let $p_i > 0$ denote the probability that a type-i object is sampled. The successive samplings are statistically independent and the sampling probabilities, $p_1, p_2, \ldots, p_N$, are fixed. When a type-i object is sampled for the first time, we say that a type-i object is detected. To find the number of samplings required for detecting a set of object types (say, object types indexed by $i = 1, \ldots, n \leq N$) is traditionally called *coupon collector problem*." In random testing, the objects are test cases U, and each type $U_i \subseteq U$ is a testing *target*: a unique failure or fault in our experiments.

Following [22], let τ_i be a random variable denoting the number of samplings required to draw a test case in target U_i, and let $\tau(n)$ be $\max\{\tau_1, \ldots, \tau_n\}$, for $n \leq N$, that is the number of samplings to cover all the first n targets (in any order). It is possible to prove that the expected value of $\tau(n)$ is:

$$\mathbb{E}\left[\tau(n)\right] = \sum_{1 \leq i \leq n} (-1)^{i+1} \sum_{J;|J|=i} \frac{1}{\sum_{j \in J} p_j} \qquad (12)$$

The inverse $\widetilde{\phi}(k)$ of $\tau(n)$ is a function from the number of test cases to the expected number of failures, hence it corresponds to an analytic version of $\phi^c_{\text{mean}}(k)$.

It is possible to approximate $\tau(n)$ for two special cases of probabilities p_i's: when they are all equal ($p_1 = p_2 = \ldots = p_N = \theta$), and when they are exponentially decreasing ($p_i = \theta/10^{i-1}$). We can show that, in the first case, $\tau(n)$ is polynomial in $\Theta(\log^h(n))$ for some constant k, hence $\widetilde{\phi}(k)$ is $\Theta(\exp(gk))$ for some constant g; in the second case, $\tau(n)$ is $\Theta(\exp(hn))$, hence $\widetilde{\phi}(k)$ is $\Theta(\log^h(k))$.

This might provide a partial explanation for why the faults in AutoTest follow a poly-logarithmic curve: the distribution of faults has exponentially decreasing probability. It may also justify some of the moderate differences in the Java experiments, if the fault distribution is different in the Java code with respect to the Eiffel code analyzed.

A more general problem related with this explanation has to do with how the coupon collector model applies to random testing of *object-oriented* programs the way it is available in AutoTest and YETI. Such tools generate test cases dynamically by incrementally populating a pool of objects with random calls. This behavior entails that the probability of sampling an object with certain characteristics (and hence of constructing a test case that belongs to a certain

target) is not fixed but dynamically varies, and successive draws are not statistically independent. For example, the first round can only successfully call creation procedures, and the probability of constructing a test case involving any other routine is zero. Hence, test cases are not really sampled uniformly at random, and the problem is to determine the connection between the real process and its representation as a coupon collection process. All we can say with the currently available data is that object-oriented random testing with AutoTest seems to be describable as a coupon collection process with a probability distribution over faults that is always exponentially decreasing; this is not quite the same as saying that the distribution of bugs is exponentially decreasing.

3.2 Threats to validity

Threats to *construct validity* are present in the Java experiments where we have no reliable way of reconstructing faults from failures; a similar threat occurs with the Eiffel experiments if the contracts incorrectly capture the designer's intended behavior. However, even if we may be measuring different things, we are arguably still measuring things that significantly correlate; the results partially confirm so.

The very large number of repeated experiments should have reduced the potential threats to *internal validity* – referring to the possibility that the study does not support the claimed findings – to the minimum [2]. As discussed in Section 2.3, however, the Java experiments are somehow less clear-cut than the Eiffel experiments; further experiments will hopefully provide conclusive evidence.

External validity – which refers to the findings' generalizability – is limited by the focus on random testing and by the availability of software that can be tested with this technique. This limitation is largely a consequence of the need of designing experiments approachable with the currently available technology. In future work, we will target more software with contracts (e.g., JML and .NET code equipped with CodeContracts) and more testing frameworks (see Section 4) to improve the generalizability of our findings.

4. RELATED WORK

Automated random testing is a technique that is inexpensive to run and proved to find bugs in different contexts, including Java libraries and applications [18, 9, 21]; Eiffel libraries [6]; and Haskell programs [7]. For brevity, we only succinctly mention the most closely related work.

Yeti and AutoTest are two representatives in a series of random testing tools developed during the last decade, including Randoop [19], JCrasher [9], Eclat [18], Jtest [20], Jartege [15], and RUTE-J [1].

Arcuri et al.'s theoretical analysis of random testing [3] is an important basis to understand also the practical and empirical side. Section 3.1 outlined the connection between the work in [3] and the present paper's, as well as the points that still remain open. Our preliminary results, hint at a plausible consistency between the two results (the theoretical and the empirical). To our knowledge, [16] is the only other empirical result about random testing; [16] considers only one model, which we have included as Φ_1 in our analysis.

Acknowledgements. We gratefully acknowledge the funding by the Swiss National Science foundation (proj. LSAT and ASII); by the Hasler foundation (proj. Mancom); by the Swiss National Supercomputing Centre (proj. s264).

5. REFERENCES

[1] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li. Tool support for randomized unit testing. In *RT*, pages 36–45. ACM, 2006.

[2] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[3] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ISSTA*. ACM, 2010.

[4] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[6] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Softw. Test., Verif. Reliab.*, 21(1):3–28, 2011.

[7] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.

[8] CMMI. `http://www.sei.cmu.edu/cmmi/`, 2011.

[9] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[10] Eiffelbase. http://freeelks.svn.sourceforge.net.

[11] Gobo. http://sourceforge.net/projects/gobo-eiffel/.

[12] S. McConnell. *Software Estimation*. Microsoft Press, 2006.

[13] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[14] G. Myers. *The Art of Software Testing*. Wiley, 1979.

[15] C. Oriat. Jartege: a tool for random generation of unit tests for Java classes. Technical Report RR-1069-I, CNRS, June 2004.

[16] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *ICST*, pages 201–210. IEEE, 2012.

[17] M. Oriol and S. Tassis. Testing .NET code with YETI. In *ICECCS*, pages 264–265, 2010.

[18] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE, 2007.

[20] Parasoft. Jtest. `http://www.parasoft.com/`.

[21] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *FASE*, volume 6603 of *LNCS*, pages 262–277, 2011.

[22] S. Shioda. Some upper and lower bounds on the coupon collector problem. *Journal of Computational and Applied Mathematics*, 200:154–167, 2007.

[23] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *ICSE*, pages 391–400. ACM, 2011.

6. APPENDIX

6.1 Experiments with Eiffel code

Table 5 displays the statistics of the comparison between Φ_5 and the other models according to their R^2 scores in the fittings across all Eiffel classes used in the experiments. The p-values characterize whether differences are statistically significant. The effect size (computed with the Z statistics as $Z/\sqrt{2N}$, where N is the number of classes, and hence $2N$ is the total sample size) characterizes the magnitude of the observed differences: an effect size of 0.1 is small; 0.3 is medium; 0.5 is large.

Table 5: Eiffel classes: differences between Φ_5 and the other Φ_i's (Wilcoxon signed-rank test).

Statistics	Φ_1	Φ_2	Φ_3	Φ_4	Φ_6	Φ_7	Φ_8	Φ_9
p-value	2.95E-03	28.46E-03	19.53E-06	1.11E-06	52.55E-09	71.79E-09	4.12E-06	61.45E-09
effect size	324.32E-03	239.05E-03	465.92E-03	531.39E-03	593.82E-03	587.73E-03	502.46E-03	590.77E-03

6.2 Experiments with Java code

Tables 6 and 7 show the complete statistics about the testing sessions with YETI on the Java classes described in Section 2.3. They are the extended versions of Tables 3 and 4.

Table 6: Java classes tested for failures: statistics per class.

Class	S	T	F	$\mathbb{E}[\sigma]$	$\mathbb{E}[\gamma]$	$\mathbb{E}[\Delta]$	$\sigma[\Delta]$
java.lang.Character	30	1.29E+05	32	9.91E-01	7.17E-01	1.24E-02	6.07E-02
java.lang.String	30	1.31E+05	84	7.58E-01	-1.34E+00	2.15E-02	1.04E-01
java.util.ArrayList	30	1.08E+05	10	4.84E-01	-1.01E+00	1.57E-03	1.85E-02
java.util.Calendar	30	1.35E+05	36	1.31E+00	4.21E-01	1.28E-02	3.89E-02
java.util.Date	30	1.24E+05	8	6.69E-02	-9.06E-01	3.31E-03	1.95E-02
java.util.HashMap	31	1.13E+05	7	2.84E-01	-3.36E+00	9.01E-04	8.89E-03
java.util.Hashtable	30	1.16E+05	14	4.42E-01	-1.45E+00	2.27E-03	2.48E-02
java.util.LinkedList	30	1.08E+05	12	6.70E-02	-1.31E+00	5.46E-03	4.55E-02
java.util.Properties	30	1.18E+05	15	1.25E+00	5.34E-01	7.16E-03	2.84E-02
java.util.SimpleTimeZone	30	1.63E+05	80	2.05E+00	-6.33E-02	1.93E-02	7.63E-02
java.util.TreeMap	30	1.07E+05	58	8.56E-01	-2.03E+00	2.63E-02	8.18E-02
Mean	30	1.23E+05	32	7.78E-01	-8.91E-01	1.03E-02	4.61E-02
Median	30	1.18E+05	15	7.58E-01	-1.01E+00	7.16E-03	3.89E-02
Stdev		1.65E+04	29	6.02E-01	1.23E+00	8.86E-03	3.07E-02

Table 7: Java classes tested for faults: statistics per class.

Class	S	T	F	$\mathbb{E}[\sigma]$	$\mathbb{E}[\gamma]$	$\mathbb{E}[\Delta]$	$\sigma[\Delta]$
java.lang.Boolean	4	2.49E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Byte	4	2.79E+05	2	4.10E-03	0.00E+00	3.00E-03	2.72E-02
java.lang.Character	4	3.67E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Class	4	2.39E+05	8	4.39E-01	1.07E-01	8.79E-03	5.34E-02
java.lang.ClassLoader	4	2.66E+05	8	2.82E-02	-3.85E-01	8.50E-03	6.36E-02
java.lang.Compiler	4	1.15E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Double	4	2.70E+05	4	2.23E-02	-1.04E-01	7.13E-03	5.93E-02
java.lang.Enum	4	4.29E+06		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Float	4	2.71E+05	3	9.52E-03	-4.40E-01	5.26E-03	5.10E-02
java.lang.InheritableThreadLocal	4	2.23E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Integer	4	3.06E+05	2	5.95E-03	1.10E-01	3.16E-03	3.43E-02
java.lang.Long	4	3.08E+05	2	8.62E-03	-2.66E-01	2.86E-03	2.66E-02
java.lang.Math	4	3.28E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Number	4	0.00E+00		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Object	4	2.23E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Package	4	3.36E+06	1	4.40E-04	0.00E+00	3.22E-04	8.96E-03
java.lang.Process	4	0.00E+00		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.ProcessBuilder	4	2.61E+05	3	1.07E-02	-4.48E-01	4.39E-03	3.80E-02
java.lang.Runtime	4	3.35E+05	10	7.92E-01	-3.78E-01	4.80E-04	1.74E-02
java.lang.RuntimePermission	4	2.52E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Short	4	2.84E+05	2	7.31E-03	-3.11E-01	2.89E-03	2.99E-02
java.lang.StackTraceElement	1	2.12E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.StrictMath	4	3.28E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.String	4	3.45E+05	4	2.91E-02	-3.32E-01	3.31E-03	3.03E-02
java.lang.StringBuffer	1	7.30E+03	4	0.00E+00	NaN	0.00E+00	2.12E-01
java.lang.StringBuilder	1	1.69E+03	6	0.00E+00	NaN	0.00E+00	2.97E-01
java.lang.ThreadLocal	3	2.21E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Throwable	3	2.38E+05		0.00E+00	NaN	0.00E+00	0.00E+00
java.lang.Void	3	0.00E+00		0.00E+00	NaN	0.00E+00	0.00E+00
Mean	4	4.68E+05	2	4.68E-02	NaN	1.73E-03	3.27E-02
Median	4	2.61E+05		0.00E+00	NaN	0.00E+00	0.00E+00
Stdev	1	9.45E+05	3	1.65E-01	NaN	2.72E-03	6.58E-02

Tables 8 and 9 show the result of fitting the models $\Phi_1, \Phi_2, \Phi_4, \Phi_5$ on the mean fault count $\phi^c_{\text{mean}}(k)$ curves, reporting the same data as for the AutoTest experiments in Section 2.2.

Tables 10 and 11 report the same statistics as Table 5 for the Java classes (failure and fault data).

Table 8: Testing of Java classes for failures: best fits with mean.

Class	Best Fit Ranking	R^2_{best}	$\text{RMSE}_{\text{best}}$	$\Delta(R^2)$	Δ(RMSE)
java.lang.Character	1 5 4 2	9.87E-01	3.49E-01	4.89E-02	4.24E-01
java.lang.String	1 5 4 2	9.94E-01	7.37E-01	3.23E-02	1.14E+00
java.util.ArrayList	2 1 5 4	9.75E-01	9.76E-02	2.01E-01	1.96E-01
java.util.Calendar	1 5 4 2	9.91E-01	5.29E-01	5.85E-03	1.56E-01
java.util.Date	2 1 5 4	9.78E-01	1.08E-01	1.14E-01	1.61E-01
java.util.HashMap	1 5 4 2	9.09E-01	1.73E-01	5.71E-02	4.78E-02
java.util.Hashtable	2 1 5 4	9.62E-01	1.44E-01	2.54E-01	2.54E-01
java.util.LinkedList	2 1 5 4	7.41E-01	4.26E-01	1.59E-01	1.15E-01
java.util.Properties	5 1 4 2	9.80E-01	2.34E-01	0.00E+00	0.00E+00
java.util.SimpleTimeZone	2 5 1 4	9.98E-01	3.81E-01	8.87E-03	5.93E-01
java.util.TreeMap	1 5 2 4	9.97E-01	5.04E-01	3.14E-03	2.05E-01
Mean	9% (ϕ_5 is best)			8.04E-02	3.00E-01
	64% (ϕ_5 is 2nd best)				

Table 9: Testing of Java classes for faults: best fits with mean.

Class	Best Fit Ranking	R^2_{best}	$\text{RMSE}_{\text{best}}$	$\Delta(R^2)$	Δ(RMSE)
java.lang.Boolean	1 2 4 5	-Inf	5.23E-10	NaN	4.01E-05
java.lang.Byte	1 5 4 2	9.01E-01	4.33E-02	1.82E-01	2.99E-02
java.lang.Character	1 2 4 5	-Inf	4.91E-12	NaN	2.41E-05
java.lang.Class	2 1 5 4	9.58E-01	1.98E-01	1.37E-02	2.91E-02
java.lang.ClassLoader	1 5 2 4	9.42E-01	1.98E-01	1.46E-01	1.73E-01
java.lang.Compiler	1 2 4 5	-Inf	2.45E-11	NaN	4.00E-05
java.lang.Double	1 5 4 2	9.28E-01	1.01E-01	1.61E-01	8.12E-02
java.lang.Enum	1 2 4 5	-Inf	2.52E-11	NaN	8.31E-06
java.lang.Float	2 1 5 4	9.80E-01	3.01E-02	1.97E-01	6.91E-02
java.lang.InheritableThreadLocal	1 2 4 5	-Inf	3.60E-12	NaN	4.27E-05
java.lang.Integer	2 1 5 4	9.85E-01	2.31E-02	3.02E-01	8.20E-02
java.lang.Long	2 1 5 4	9.85E-01	2.10E-02	2.10E-01	6.05E-02
java.lang.Math	1 2 4 5	-Inf	6.39E-11	NaN	4.07E-05
java.lang.Number	1 2 4 5	NaN	0.00E+00	NaN	1.18E-04
java.lang.Object	1 2 4 5	-Inf	3.41E-12	NaN	3.60E-05
java.lang.Package	2 1 5 4	9.94E-01	1.84E-03	2.16E-01	9.73E-03
java.lang.Process	1 2 4 5	NaN	0.00E+00	NaN	1.30E-04
java.lang.ProcessBuilder	1 5 4 2	9.72E-01	4.48E-02	1.22E-01	5.91E-02
java.lang.Runtime	5 1 4 2	8.34E-01	1.35E-01	0.00E+00	0.00E+00
java.lang.RuntimePermission	1 2 4 5	-Inf	1.63E-11	NaN	3.18E-05
java.lang.Short	2 1 5 4	9.94E-01	1.25E-02	2.60E-01	7.12E-02
java.lang.StackTraceElement	1 2 4 5	-Inf	4.44E-10	NaN	2.05E-05
java.lang.StrictMath	1 2 4 5	-Inf	2.20E-10	NaN	2.82E-05
java.lang.String	1 5 4 2	9.72E-01	6.20E-02	9.72E-02	7.00E-02
java.lang.StringBuffer	2 1 5 4	1.00E+00	3.58E-02	1.08E-03	3.34E-02
java.lang.StringBuilder	2 5 1 4	9.91E-01	7.40E-02	1.34E-03	4.87E-03
java.lang.ThreadLocal	1 2 4 5	-Inf	5.17E-12	NaN	4.28E-05
java.lang.Throwable	1 2 4 5	-Inf	2.88E-12	NaN	1.71E-05
java.lang.Void	1 2 4 5	NaN	0.00E+00	NaN	1.59E-04
Mean	3% (ϕ_5 is best)			NaN	2.67E-02
	24% (ϕ_5 is 2nd best)				

Table 10: Java failures: differences between Φ_5 and some other Φ_i's (Wilcoxon signed-rank tests).

Statistics	Φ_1	Φ_2	Φ_4
p-value	18.55E-03	123.05E-03	976.56E-06
effect size	492.85E-03	341.21E-03	625.54E-03

Table 11: Java faults: differences between Φ_5 and some other Φ_i's (Wilcoxon signed-rank tests).

Statistics	Φ_1	Φ_2	Φ_4
p-value	20.26E-03	855.22E-03	122.07E-06
effect size	300.87E-03	28.85E-03	432.75E-03

6.3 Eiffel experiments: all graphs

Figures 1–7 display, for each of the 42 Eiffel classes tested, the mean curve (in black) and the three models that fit best. Horizontal axes are scaled by millions of test cases drawn; vertical axes by total number of faults found.

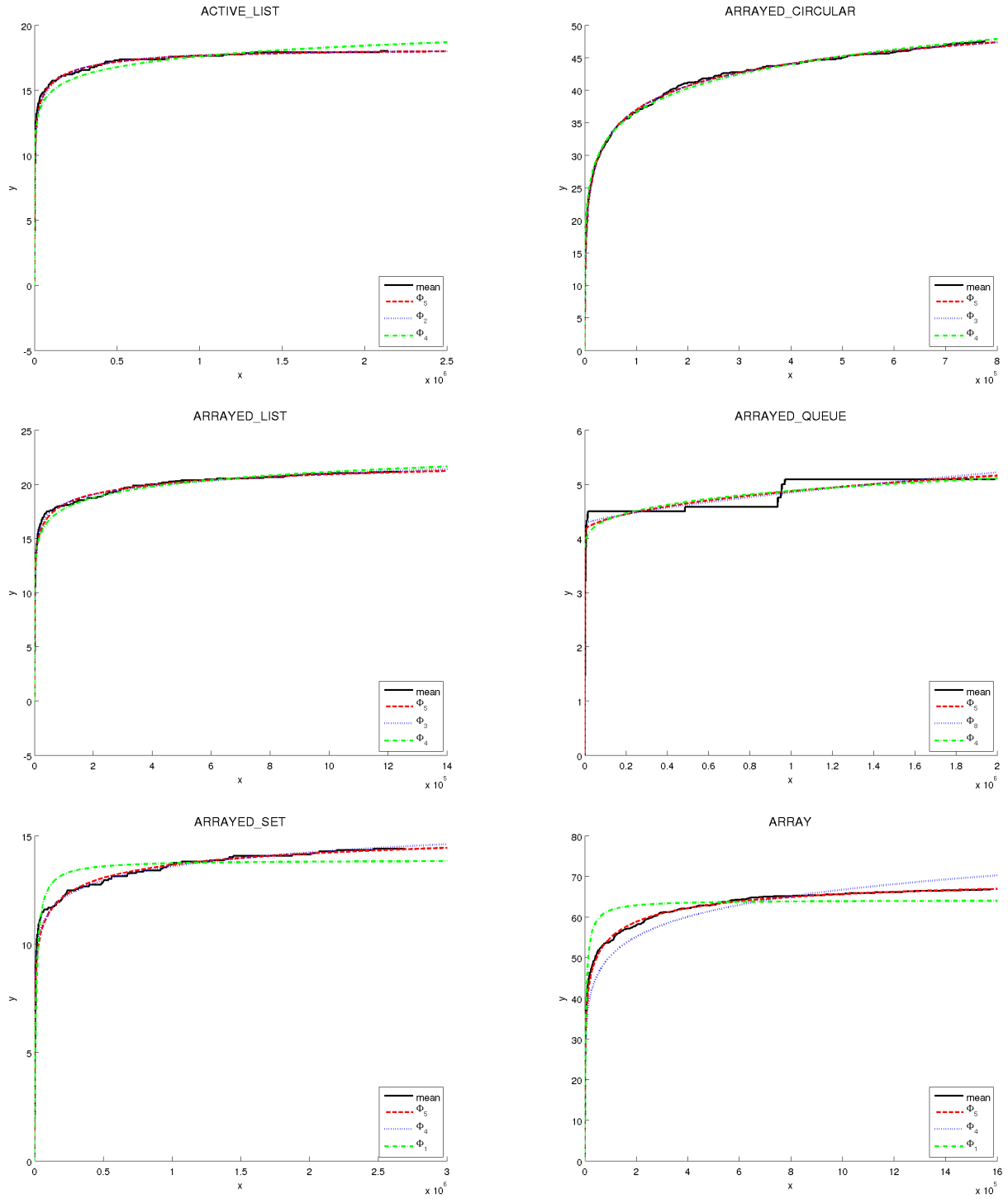

Figure 1: Top 3 fits with mean for six Eiffel classes.

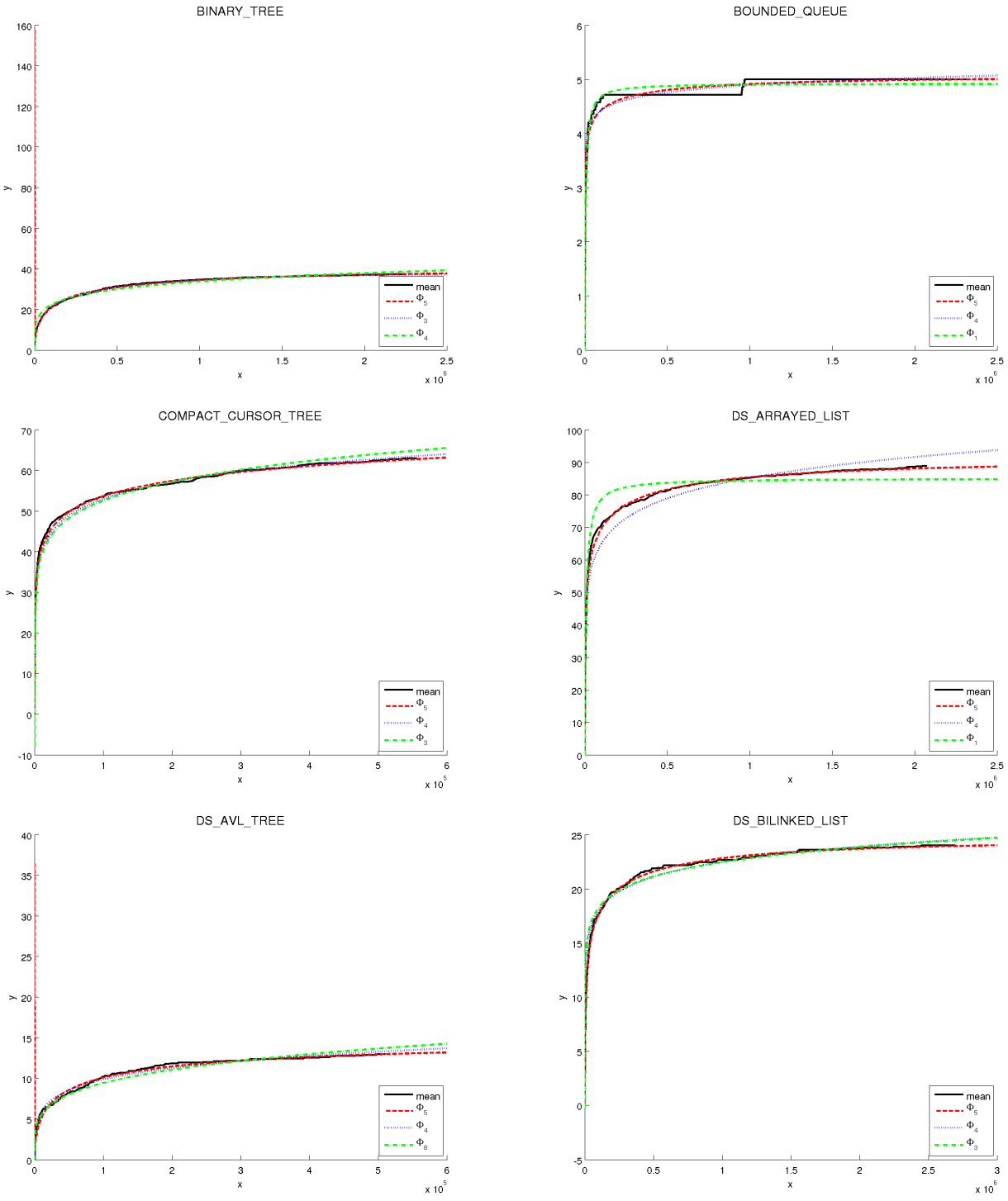

Figure 2: Top 3 fits with mean for six Eiffel classes.

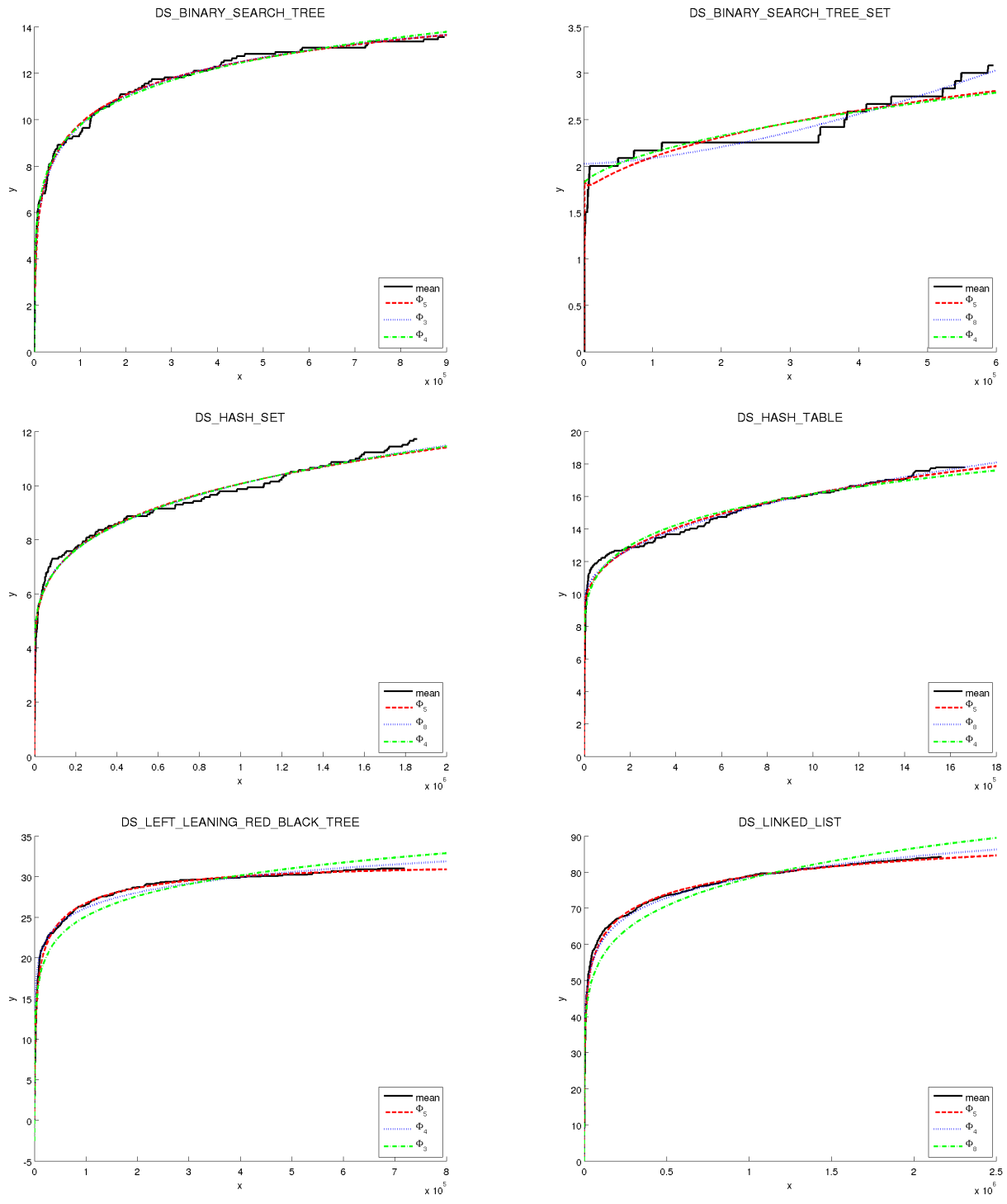

Figure 3: Top 3 fits with mean for six Eiffel classes.

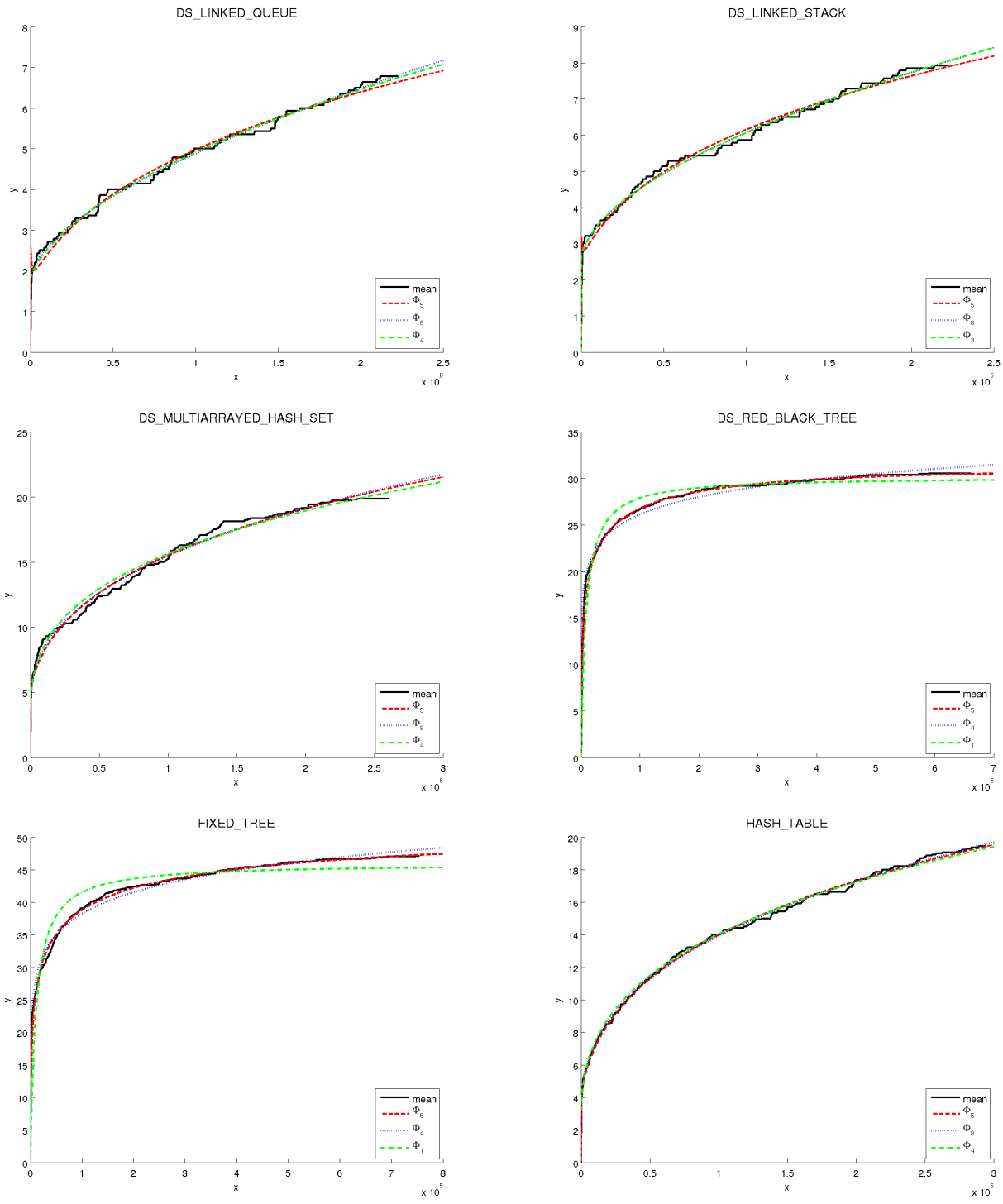

Figure 4: Top 3 fits with mean for six Eiffel classes.

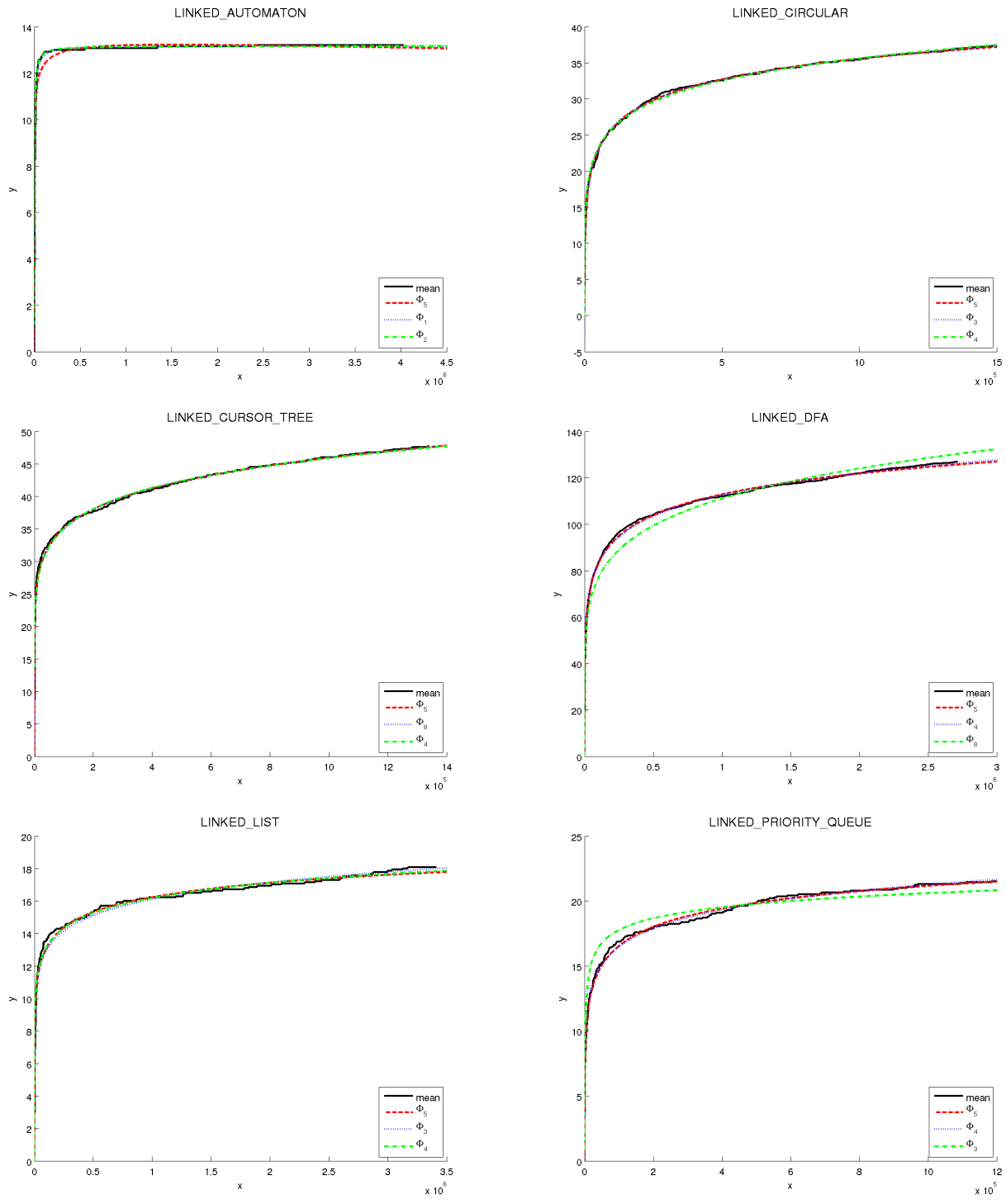

Figure 5: Top 3 fits with mean for six Eiffel classes.

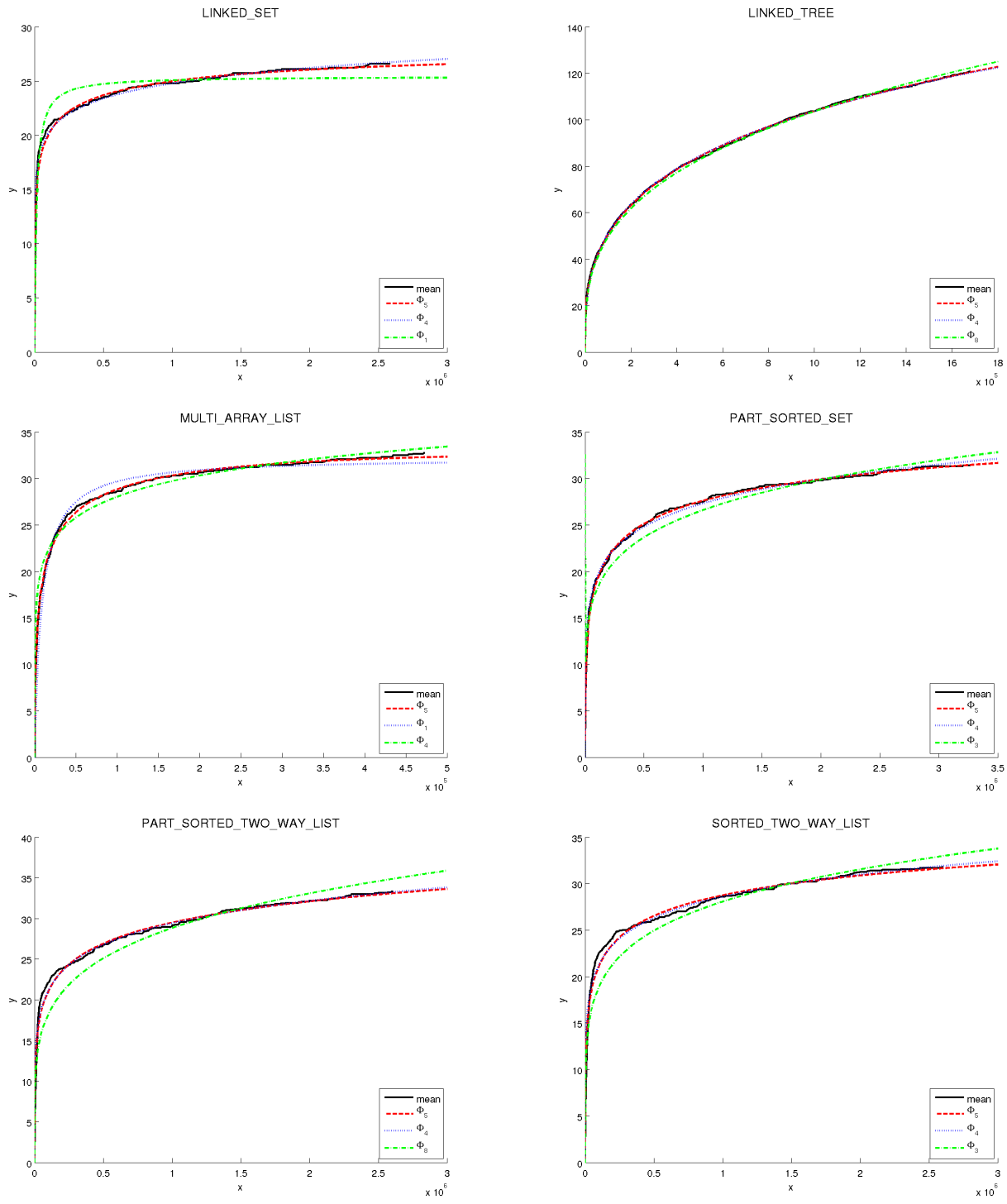

Figure 6: Top 3 fits with mean for six Eiffel classes.

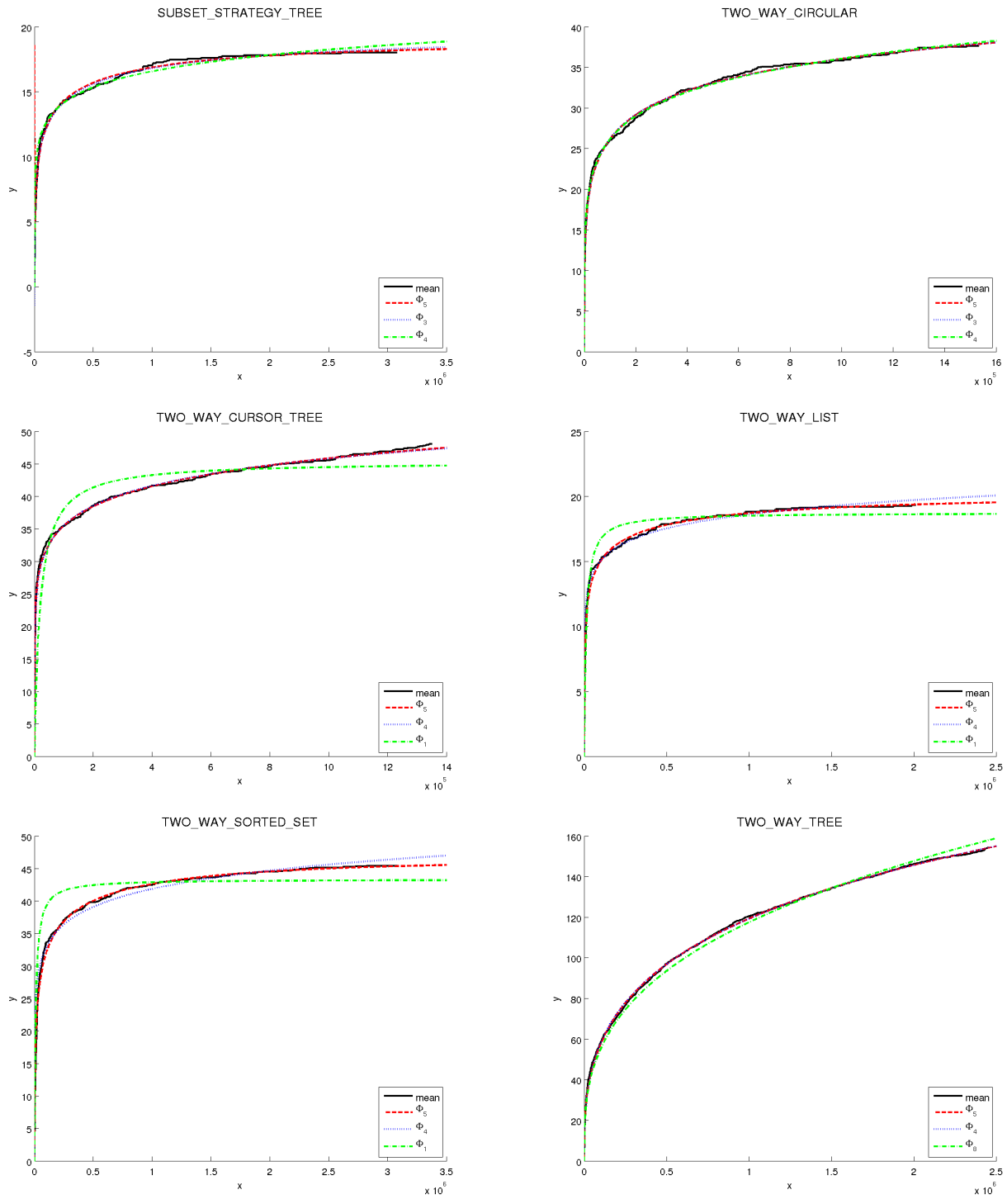

Figure 7: Top 3 fits with mean for six Eiffel classes.

6.4 Java failure experiments: all graphs

Figures 8–9 display, for each of the 11 Java classes tested for failures, the mean curve (in black) and the three models that fit best. Horizontal axes are scaled by tens of thousands of test cases drawn; vertical axes by total number of failures found.

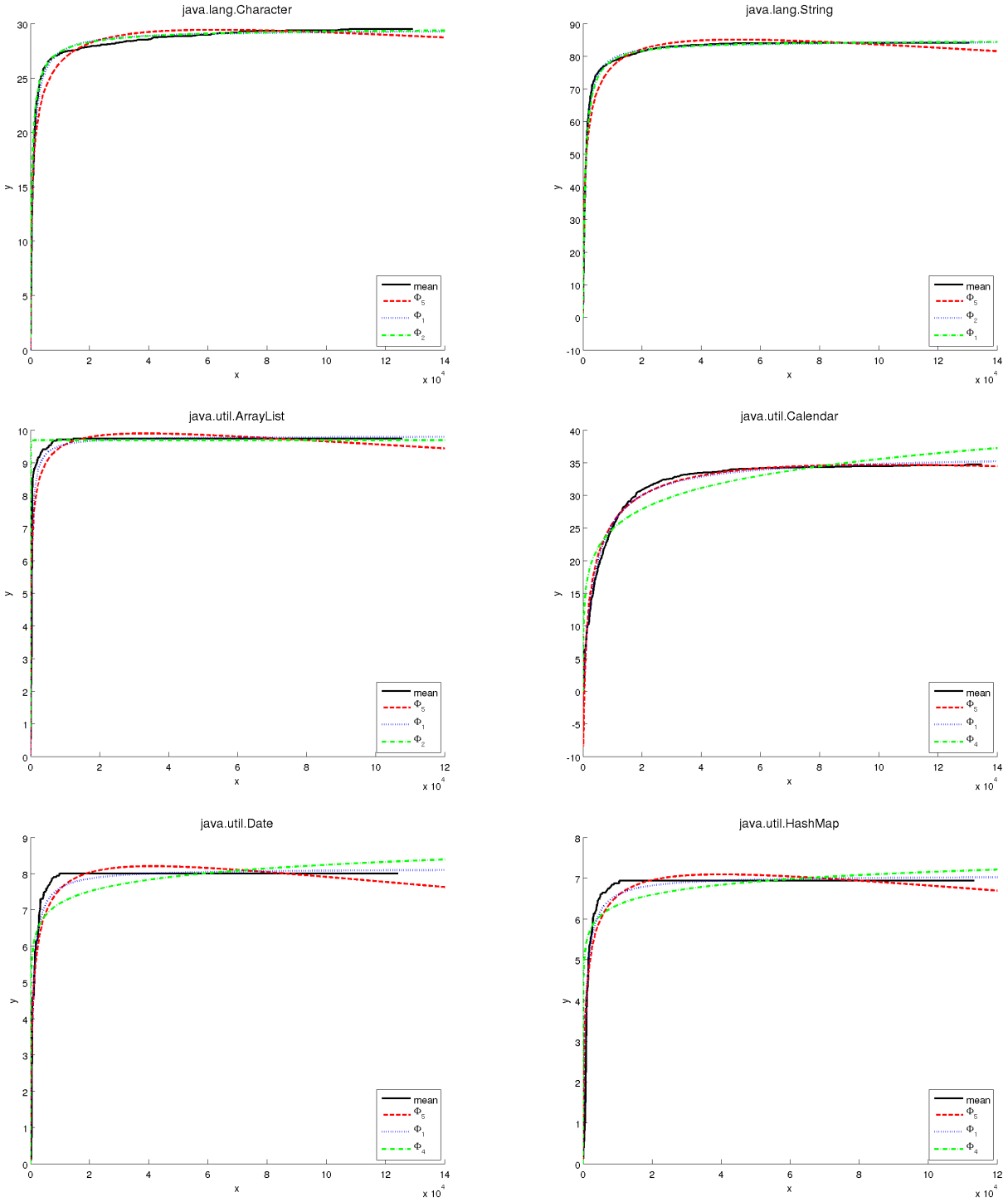

Figure 8: Top 3 fits with mean for six Java classes (failures).

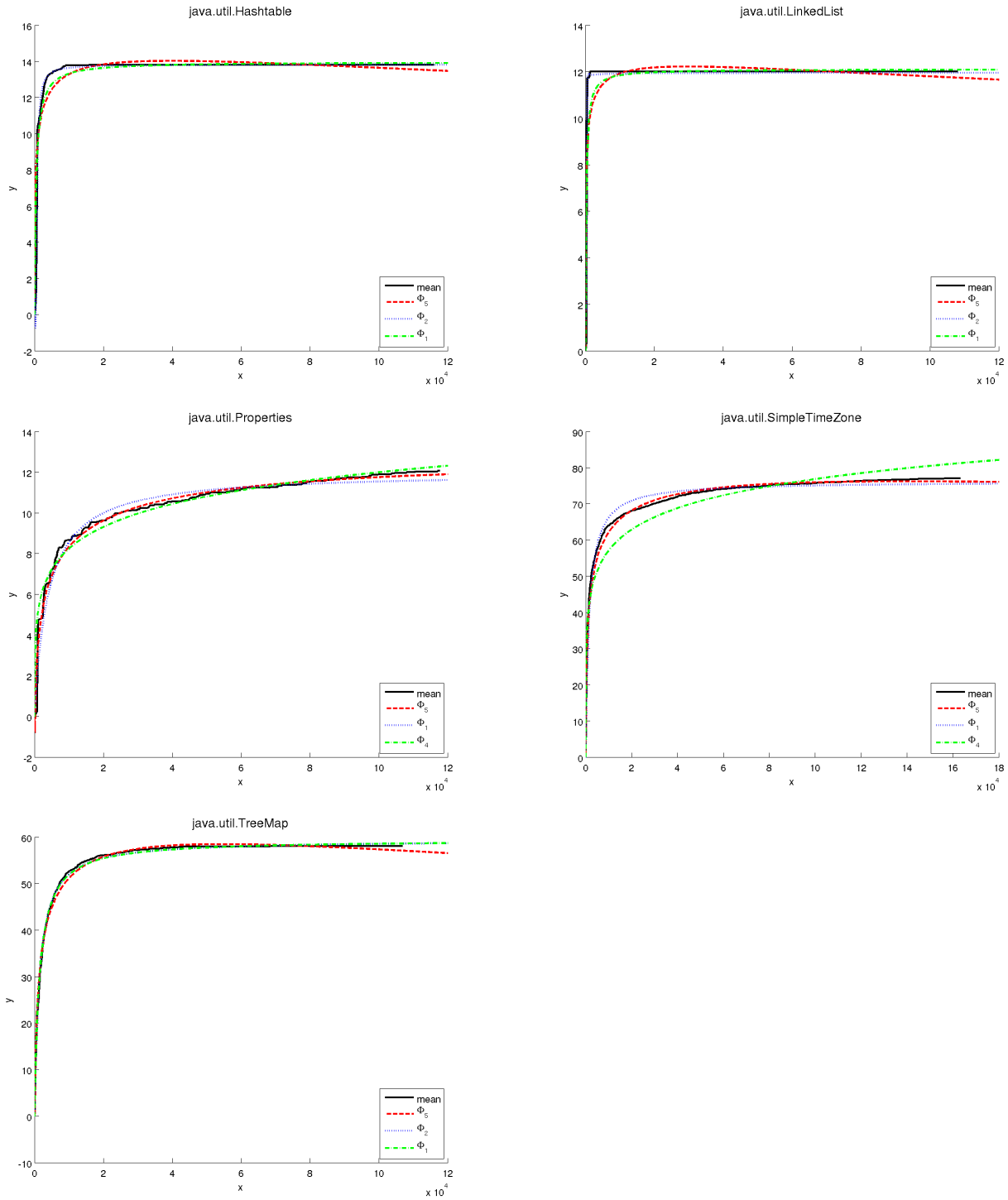

Figure 9: Top 3 fits with mean for five Java classes (failures).

6.5 Java fault experiments: all graphs

Figures 10–12 display, for each of the 29 Java classes tested for faults where at least one fault was found, the mean curve (in black) and the three models that fit best. Horizontal axes are scaled by hundreds of thousands of test cases drawn (except for class *java.lang.StringBuilder* which is not scaled); vertical axes by total number of faults found. The last values of the mean curve for *StringBuffer* and *StringBuilder* are measurement errors that should be ignored.

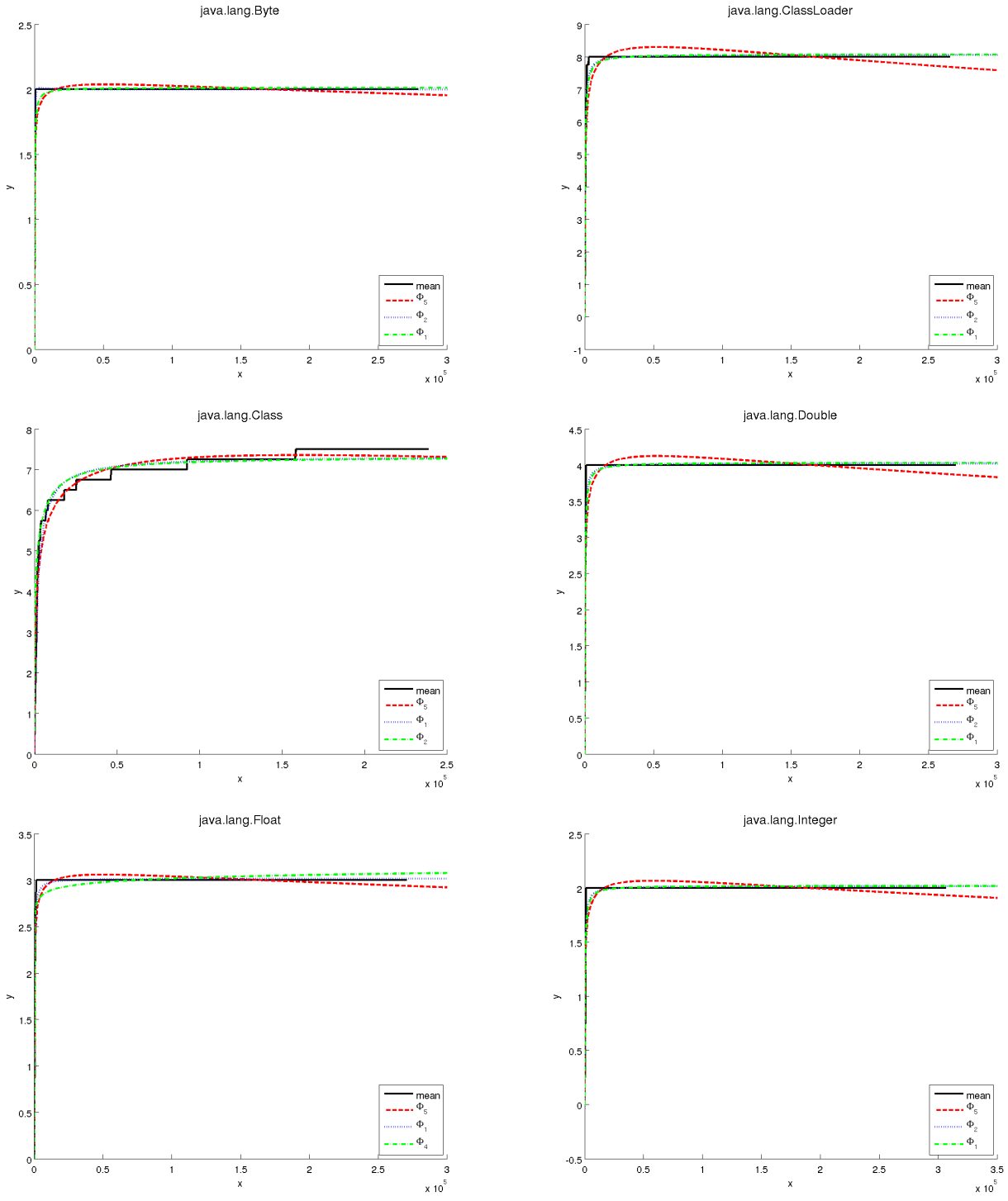

Figure 10: Top 3 fits with mean for six Java classes (faults).

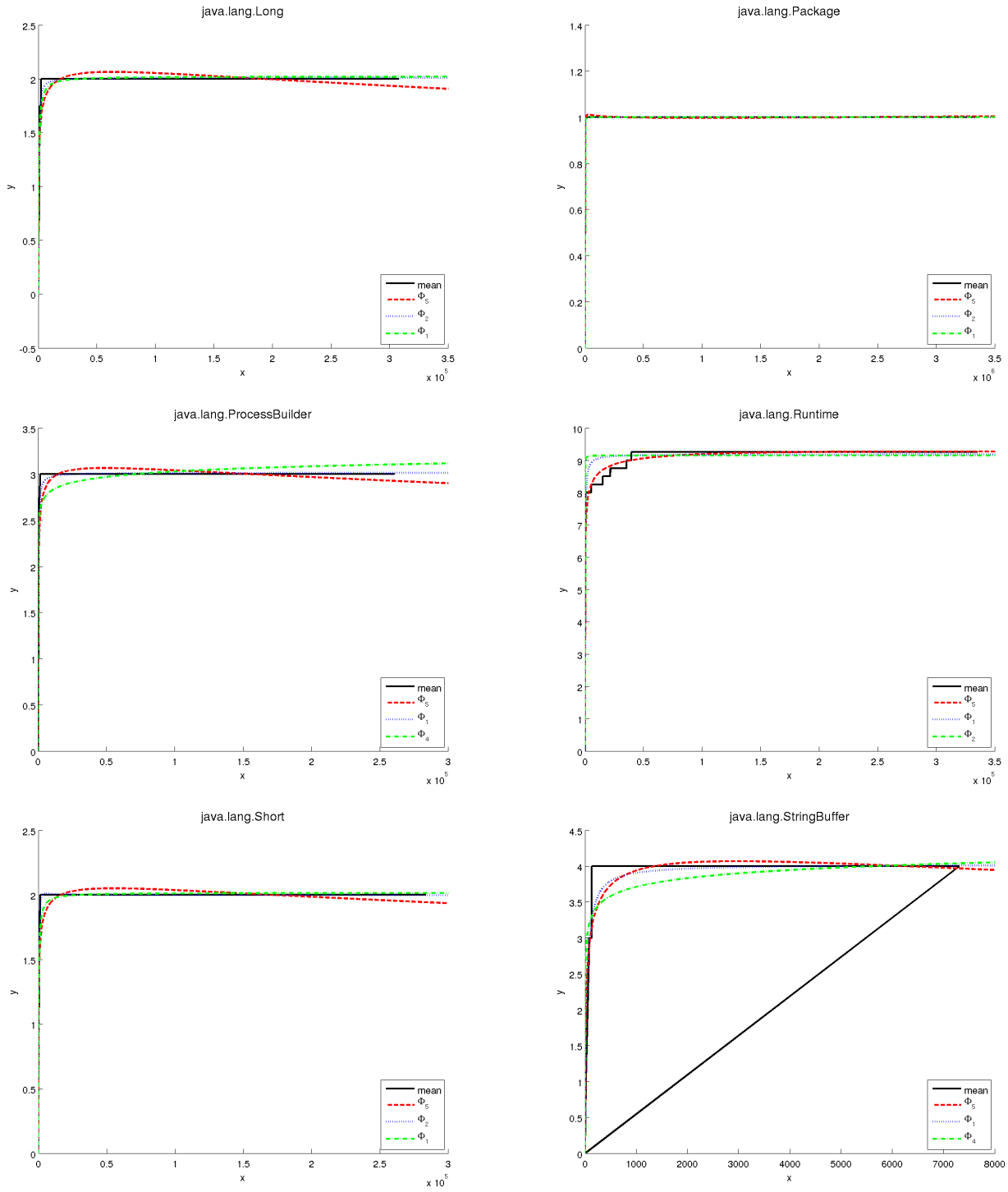

Figure 11: Top 3 fits with mean for six Java classes (faults).

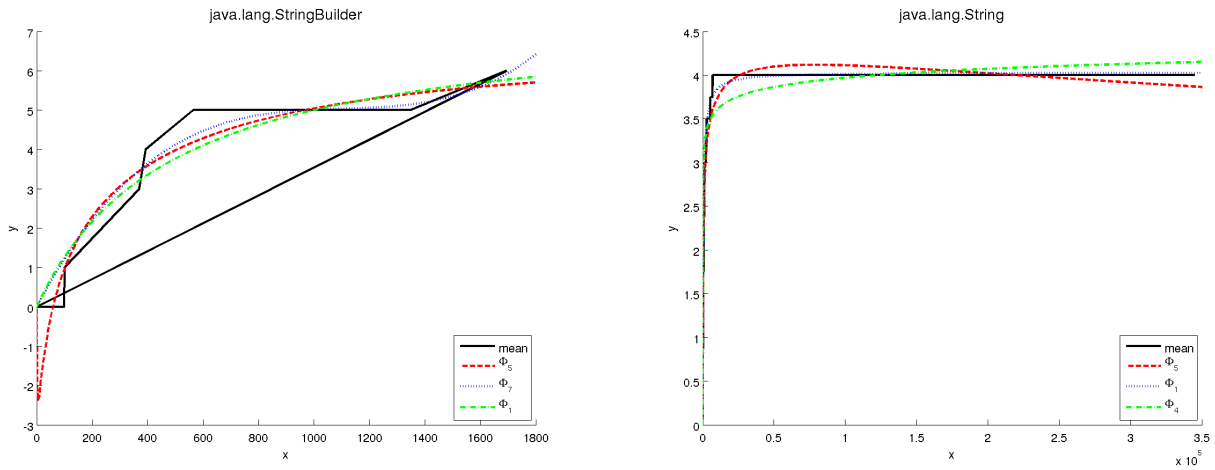

Figure 12: Top 3 fits with mean for two Java classes (faults).